Initial Report

Team Name: Gossip-4
Team Members: Wladislaw Meixner, Robert Schambach
Module: Gossip

Introduction

We are team **Gossip-4** and intend on implementing the **Gossip** module. Our team is composed of Wladislaw Meixner and Robert Schambach.
Wladislaw Meixner is well versed in multiple programming languages, especially golang, TypeScript, Swift, Python and others. With multiple years of golang server and typescript frontend development experience in the ecommerce market. Additionally, Meixner has proficient experience in server maintenance, Docker, Kubernetes setup as well as other dev ops tasks.
Robert Schambach is an experienced software engineer who is currently working in the IT-Security domain. Schambach has a well-rounded technical background; he has executed many projects in many languages, including Rust, Bash, Java, and Python. Further, Schambach is proficient in integration and deployment of software systems, using technologies such as Docker, Docker-Compose, and Kubernetes.

Project overview

In this section, we describe the sketch of our Gossip module. We initially discuss the internal structure of the module. Further, we state the license under which we distribute the software of this module.

Design

The proposed design implements the Gossip module. As such, our project structure can be split into five key submodules: P2P-Communication, API-Communication, Proof-of-Work, Validator, Publisher, and Broadcaster.
The communication submodules, P2P- and API-Communication, define all message types listed in the project specification, with matching serialization and deserialization procedures.
Furthermore, they expose an abstraction layer that other submodules can use for reliable communication with peers and modules. In particular, the P2P-Communication submodule handles communication with Gossip modules on other peers, while the API-Communication submodules addresses communication with other modules on the same host. These submodules also forward incoming messages, once serialized, to the corresponding Gossip submodule. The P2P-Communication protocol is a topic of ongoing discussion in our group, the current consensus is that we at least require the following fields in our messages: data type, TTL, size, Sender ID, Target ID, nonce and data. These messages will be coded using Protocol Buffers. We intend to use UDP to spread information between sibling Gossip instances.
The Broadcaster submodule is responsible for retransmitting valid P2P-Communications as well as host originated `GOSSIP ANNOUNCE` messages to other peers on the network. External P2P-Communications are forwarded to the Proof-of-Work and Validator submodules to ensure validity before further propagating the contained information to the host and other peers. Further, the Broadcaster uses the communication submodules to communicate with the RPS module and other Broadcaster instances on sibling peers. The RPS module is utilized to provide the Broadcaster with a list of known peers for information spreading. Additionally, this submodule maintains a list of to be propagated data items, called knowledge base. The capacity of this list is determined by the `cache_size` parameter, defined in the module configuration. We intend to utilize a Ring Buffer like structure to overwrite older knowledge base items if the `cache_size` is exceeded. Given the buffer is not overfilling, items are kept in the knowledge base as long as they are not transmitted to `degree` number of peers, as specified in the module configuration. If the RPS module does not provide the Broadcaster with a sufficient number of peers, periodic RPS queries will be performed in order to acquire new hosts and send out the remaining items in the knowledge base.

Complementary, the Proof-of-Work module provides hash computation and validation primitives used by other submodules. Our proposed Proof-of-Work algorithm is analogous to the one required in the registration part of the assignment. Additionally, we want to link the TTL to the difficulty of our Proof-of-Work computation, with larger TTLs

requiring a more time-consuming computation.

The Publisher submodule handles subscriptions to the Gossip module. Other submodules may subscribe to the Gossip module by sending `GOSSIP NOTIFY` messages. The API-Communication submodules pass these messages to the Publisher submodule. The Publisher thereby registers a subscription for messages of the type contained in the `GOSSIP NOTIFY` message. If the Publisher then receives messages of the corresponding message type, the submodule wraps the messages in a `GOSSIP NOTIFICATION`. Moreover, the Publisher will attempt to send the notification message over the API-Communication submodule. If the sending fails, as e.g., the connection has been closed by the receiving module, the Publisher registers the connection loss as an unsubscribe. Otherwise, the Publisher expects a `GOSSIP VALIDATION` message, which it returns to its caller.
The Validator submodule's responsibility is to validate the message contained in incoming broadcast messages sent by other peers by communicating with other internal modules. The Validator thereby sends the to-be-validated message to subscribed internal modules over the Publisher submodule. Afterwards, the Publisher submodule returns a `GOSSIP VALIDATION` message from the subscribed submodule. The Validator then asserts the to-be-validated message as valid if the bit field `v` of the `GOSSIP VALIDATION` message is set, i.e., if the message contained in the broadcast message is well-formed. Further, if Validator does not receive a response message, as e.g. no other module subscribes to the contained message type, Validator asserts that the message is invalid as well.

Licensing

We aim to offer our project to others as a reference to supply value beyond the scope of this university course. Therefore, it is our intention to allow for equally open access and distribution of the source code and all of its artifacts. As such, we intend to publish all results under the permissive MIT-0 license [19]. In contrast to the MIT [20] license, the MIT-0 omits the attribution clause. This omission allows for unlimited sublicensing. Furthermore, we have made sure to select libraries that are at least as permissive as the MIT licensing agreement to match our goals.

Implementation

In the following section, we will discuss the aspects relevant for the realization of the Gossip module. These aspects include the selected programming language, libraries, build system and supported platforms.

Programming Language

Rust was chosen as the primary programming language for this project. This particular language was chosen for its high flexibility, performance, and memory safety. Additionally, Rust supplies useful built-in tooling, such as Cargo and rustfmt [16]. Moreover, the team members are partially acquainted with Rust.

Supported Platforms

The Rust programming language supports a multitude of operating systems and instruction set architectures out-of-the-box [1]. Further, during implementation, we take care to use operating-system agnostic concepts, e.g. fixed Unix file paths. As such, cross compilation for many platforms is easily achieved.
Yet, as we implement and test the module on Unix-based platforms, we primarily support such platforms. We do not guarantee the software to function as intended on other platforms, e.g. on Windows operating systems.

Build

Rust offers its own build system and package manager, namely Cargo [17]. We use Cargo to build, manage dependencies and potentially cross compile our Gossip module. Depending on the final project structure, it may be required to split our project into multiple independent submodules, which in terms requires further build tooling. We intend to use a combination of Cargo workspaces and Cargo scripts. Cargo workspaces aid with building and organizing of multiple packages, library crates in rust terminology, in a single monorepo project [11]. Additionally, we intend to use Cargo Build-Scripts for minor tasks such as Protocol Buffer generation.

Libraries

As Gossip is a networking module, we require libraries suitable for networking.
To this end, we use tokio [2], an asynchronous runtime for Rust. This library enables our module to use network functions in an asynchronous manner, offering substantial performance benefits without compromising the application's reliability, flexibility, or simplicity.
We encode the P2P-Gossip messages using Protocol Buffers [5]. Furthermore, we utilize Protocol Buffers in Rust using rust-protobuf [6]. We chose this encoding as it is simple to

write, maintain, and use.
To counter common peer-to-peer network attacks such as Eclipse and Sybill, our Gossupe module intends to use a Proof-of-Work-based peer identity validation schema, requiring peers to calculate a hash with certain properties [7]. As these operations are computationally intensive, we choose the performant libraries ring [8] and rand [9]. Ring contains optimized functions for computing a suitable hash, and rand allows for fast generation of pseudo-random numbers.
For error handling, we primarily use the Rust standard error library. To avoid writing boilerplate code for error handling, we use the thiserror [3] library.
We parse the user-specified Windows INI formatted configuration file for Gossip using the ini [10] library.

Quality Assurance

A structured software development workflow promotes high code quality [18]. As we host our code in GitLab, we decide to use GitLab flows [14]. We want to adopt a merge request based GitLab workflow to keep the team in sync and catch problems early. After the team approval, GitLab pipelines are used to test functionality and quality. For testing these software attributes, we intend to use two pipelines, namely: Lint-Build-Test and End-To-End pipelines [15].
The Lint-Build-Test pipeline, as the name suggests, performs basic linting, Cargo check, builds the module and runs Cargo tests. Additionally, we intend to use LLVM coverage tools [12] in this pipeline to generate test and documentation coverage reports. GitLab runs the pipeline on all open merge requests; this process must succeed before a given PR can be merged into the main branch.
The End-To-End pipeline will be added later during the development process to test the Gossip module in its entirety and thereby ensure basic functionality and interoperability with other modules. As this pipeline is assumed to be slow, we only intend to run E2E tests on the main branch.
To ensure consistent code style, we intend to use pre commit hooks in combination with Cargo Husky [21] allowing us to vent code style and linting issues before a commit can be pushed to git.
Documentation is the final quality assurance measure included in this project. Rustdoc, a rust native documentation generation tool, is selected for this task. This approach allows us to write documentation in tandem with source code and enables automated documentation coverage tests. A key advantage of including documentation in the source code is that documentation and implementation are continuously kept in sync.

Workload distribution

The segments of the Gossip module described in the Design section provide a baseline for workload subdivision. We expect the Publisher and Broadcaster submodules to require the most effort, as they contain the majority of the module specific logic. As the remaining submodules are limited in complexity and scope, we estimate these submodules to be easier to implement. As such we have distributed the workload as follows:

Schambach's workload:

- Publisher
- Validator
- API-Communication

Meixner's workload:

- Broadcaster
- P2P Communication
- Proof-of-Work

References

[1] "Platform Support - The rustc book." https://doc.rust-lang.org/nightly/rustc/platform-support.html (accessed May 28, 2022).
[2] "Tokio - An asynchronous Rust runtime." https://tokio.rs/ (accessed May 28, 2022).
[3] "thiserror - Rust" https://docs.rs/thiserror (accessed May 28, 2022).
[4] "sled - Rust." https://docs.rs/sled/ (accessed May 28, 2022).
[5] "Protocol Buffers | Google Developers." https://developers.google.com/protocol-buffers (accessed May 28, 2022).
[6] S. Koltsov, rust-protobuf. 2022. Accessed: May 28, 2022. [Online]. Available: https://github.com/stepancheg/rust-protobuf
[7] G. Carle, "Attacks in P2P Systems" https://www.moodle.tum.de/pluginfile.php/3711620/mod_resource/content/1/05_P2P_Attacks.pdf (accessed May 19, 2022).
[8] B. Smith, ring. 2022. Accessed: May 28, 2022. [Online]. Available: https://github.com/briansmith/ring
[9] Rand. rust-random, 2022. Accessed: May 28, 2022. [Online]. Available: https://github.com/rust-random/rand

[10] “ini - Rust.” https://docs.rs/ini/1.3.0/ini/ (accessed May 28, 2022).
[11] “Cargo Workspaces” https://doc.rust-lang.org/book/ch14-03-Cargo-workspaces.html (accessed May 29, 2022)
[12] “Instrumentation Coverage” https://doc.rust-lang.org/nightly/rustc/instrument-coverage.html (accessed May 29, 2022)
[13] G. Carle, “Random Peer Sampling” https://www.moodle.tum.de/pluginfile.php/3715349/mod_resource/content/1/06_RPS.pdf (accessed May 19, 2022).
[14] “Gitlab Flow” https://docs.gitlab.com/ee/topics/gitlab_flow.html (accessed May 30, 2022)
[15] “CI/CD Pipelines” https://docs.gitlab.com/ee/ci/pipelines/ (accessed May 30, 2022)
[16] “Rust - Foreword” https://doc.rust-lang.org/book/foreword.html (accessed May 30, 2022)
[17] “Cargo” https://doc.rust-lang.org/Cargo/reference/specifying-dependencies.html (accessed May 30, 2022)
[18] “State of DevOps Report 2021” https://cloud.google.com/devops/state-of-devops (accessed May 30, 2022)
[19] “MIT-0 License” https://github.com/aws/mit-0 (accessed May 30, 2022)
[20] “MIT License” https://github.com/git/git-scm.com/blob/main/MIT-LICENSE.txt (accessed May 30, 2022)
[21] “Cargo Husky” https://github.com/rhysd/cargo-husky (accessed May 30, 2022)